VMM Verification Methodology Manual 活用テクニック

デバイスの記事

赤星博輝

第2回　テストベンチの作成にVMMの部品を利用する

検証ライブラリとその利用ガイドラインである“Verification Methodology Manual for System Verilog（VMM）”の活用法を解説する連載の第２回である．今回は実際にVMMの部品を用いてテストベンチを作成する．また，vmm_data，vmm_channel，vmm_atomic_gen，vmm_xactor，vmm_envの使いかたを説明する．検証のエキスパートが作成したVMMのエッセンスを理解することで，検証の再利用性や効率を引き上げることができる．（編集部）

前回は，SystemVerilogのオブジェクト指向に関する部分と，VMMを使う際にいろいろな場面で出てくるvmm_logについて説明しました．今回は，簡単な設計に対してVMMを使ってテストベンチを作成してみます．

VMMでテストベンチを作成するため，今回は五つの部品，すなわちデータを扱うvmm_data，テストベンチ間でデータのやり取りを行うvmm_channel，ランダム・パターン生成（以下，ランダム生成）を行うvmm_atomic_gen，処理を記述するvmm_xactor，テストベンチ環境を構築するvmm_envについて紹介します．

ところで，なぜVMMにはいろいろな構成要素があるのでしょうか？ それはテストベンチを効率的に設計するためです．前回説明したvmm_logを毎回作ると，かなりの工数を必要としますし，再利用しやすく作るのはさらにたいへんです．日常の検証作業で，次の開発プロジェクトのことを考えて行動する人はどのくらいいるのでしょうか？

VMMでは基本となる部品をあらかじめ定義しており，その部品をカスタマイズして使うという手法をとります．ただし，カスタマイズするときにコードをカット・アンド・ペーストして変更を加えると，だんだんわけがわからなくなってきます．VMMでは，オブジェクト指向言語の

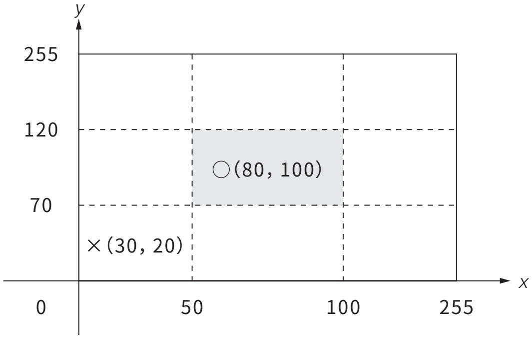


図1　ターゲットの判定回路の動き

256×256の中に(50，70)～(100，120)の長方形があり，その領域内であれば‘1’，領域外であれば‘0’と判定する．

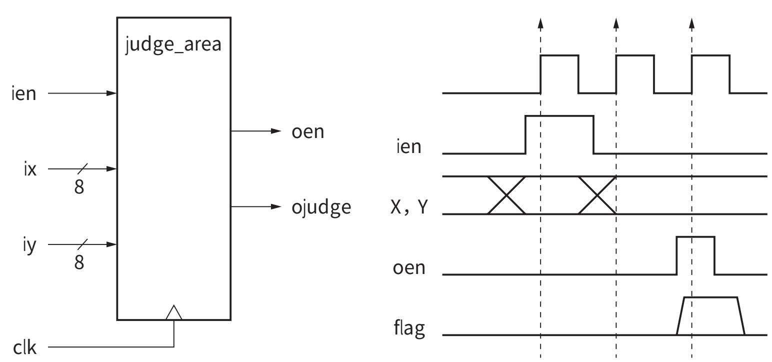


図2　ターゲットの入出力とそのタイミング

データを入力するときはienを‘1’にし，出力するときはoenが‘1’になる．

KeyWord　SystemVerilog，VMM，vmm_log，vmm_data，vmm_channel，vmm_atomic_gen，vmm_xactor，vmm_env，ランダム・パターン生成，テストベンチ

機能を使うことで，変更点だけを追加・変更できるようになっており，検証の再利用を促進します（下掲のコラム「VMMの歴史」を参照）．

● 今回の検証ターゲットはx，y平面の判定回路

VMMは，検証するための道具とそのガイドです．検証するものがないと説明しにくいので，検証のターゲットとしてここでは単純な判定回路を使用します．その回路とは，**図1**に示す領域に対して，「入力された点（x，y）が四角形の中にあるのか，または外にあるのか」を判定するものです．回路の入出力とデータの入力/出力タイミングを**図2**に示します．

この回路を検証するためにどのようなデータが必要かを考えます．機能としては*X*座標と*Y*座標を与えて領域外/領域内の判定を行うものなので，*X*座標と*Y*座標の二つの数値が必要です．ほんとうは`ien`も必要なのですが，これはあとで考えます．

リスト1　データのクラスの基本
メッセージ・サービス（`vmm_log`），変数，`new`の三つが最低限必要．

```
class xy_dat extends vmm_data;

  static vmm_log log=new("XY_dat", "class");

  rand logic[7:0] mX,mY;

 function new( );
  super.new(log);
 endfunction

 //作成したほうがよいメソッドはここに置く

endclass
```

- データのクラスはvmm_dataを派生させて作る
- メッセージ・サービスをクラスに一つ宣言する
- データを保存する変数mX，mYをrand属性で定義する
- 領域が確保されたときに最初にやるべき項目を記述する

● 検証で扱うデータはすべてvmm_dataから派生させる

VMMでは，検証で扱うデータを`vmm_data`から派生させて作ります．今回は，二つの数値を記憶するために，`vmm_data`から派生させた`xy_dat`というクラスを作ります．この`vmm_data`から派生したクラスを構成するうえで最低限必要なものは，**リスト1**に示すように`vmm_log`，必要な変数の定義，`new`メソッドの三つになります．

VMMのメリットはこれまでの検証の知識がその部品の中に埋め込まれているところで，vmm_logはVMMの「Rule 4-58」にガイドとして示されているように，静的（static）なメッセージ・サービスとして宣言します．これは，多くのデータが処理されるときに，個別にメッセージ・サービスを作ると負荷が重いことなどが理由です．

必要な変数を定義するため，今回必要となるデータの詳

コラム　VMMの歴史

VMMの歴史は，著者のひとりであるJanick Bergeron氏の検証への取り組みの歴史ともいえます．Janick Bergeron氏からVMMの歴史について紹介していただいたので，同氏のコメントを以下に紹介します．

「まずは1991年にセルフ・チェック機構をもつ，トランザクション・レベルのテストベンチを使い始めました．この段階ではVerilog HDLでモデルを記述し，第1世代の`vmm_log`，`vmm_data`，`vmm_env`相当のものが作成されました．このときはダイレクト・テストで物理レベルのトランザクタを使用し，バックグラウンドのノイズにはランダム生成も使いました．1994年にSONET（synchronous optical network）伝送の検証でより高い抽象度のトランザクタが必要となったのですが，Verilog HDLでは不十分なものしか記述できませんでした．

その後，Veraやe言語の登場によりオブジェクト指向的な実装が可能となり，1998年に第1世代の`vmm_channel`を作成しました．

2001年に（Janick氏がChief Technical Officerを務めていた）米国Qualis Design社が検証IPを提供したときは，すべてのトランザクタが`vmm_xactor`をベースにしており，`vmm_log`もほぼ現在の形になっていました．トランザクタもコールバック・メソッドを提供するようになり，ユーザによる機能の拡張や修正が可能となりました．VMMで重要なランダム生成におけるファクトリ・パターンも実装されました．

2002年の終わりには，ほぼ今の形の`vmm_env`を作りました．これは，検証環境を構築するために明確なガイドラインが欲しいということから作成しました．ほかにも，顧客からの要求に基づいて`vmm_broadcast`や`vmm_scheduler`などが作成されました．

2003年に米国Synopsys社に所属することになり（Qualis Design社がSynopsysに買収された），VeraでRVM（Reference Verification Methodology）の実装を行い，`vmm_notify`，`vmm_atomic_gen`，`vmm_scenario_gen`などを追加していきました．2004年にVeraのRVMをSystemVerilogで実装し直し，2005年にVMMを書籍として出版しました」．

このように，Janick Bergeron氏は15年以上もの間，検証にかかわってきました．検証に対して積極的に環境を改善し，その活動の集大成がVMMということになります．とても参考になる部分が多いので，みなさんもぜひ，その知識に触れていただきたいと思います．

細を検討します．入力するデータは8ビットで符号なしの整数になります．そのため，logic[7:0]の変数を定義すればよいのですが，VMMではデータ型について重要なツールがあります．

VMMの「Rule 4-59」には，「変数はrand属性をつけて定義する」とあります．これは，rand属性はOFFすることはできるのですが，あとからrand属性を付けることはできないためです．例えば，そのデータの意味から本来はランダム生成しない変数でも，わざとエラーを発生させたりするためにランダム生成を使用したくなる場合があります．CRC（cyclic redundancy check）などが良い例で，ほかのデータによってCRCの値は決定されるので，本来はランダム生成を使用しないのですが，エラー状態をシミュレーションするためにCRCの値をランダム生成したい場合が出てくることがあります．

このような知識が書籍全体にちりばめられているところがVMMのおもしろいところです．

● 単体テストでは，まず四つのメソッドを作成

メッセージ・サービスvmm_logをスーパクラスのnewメソッドに渡す必要があるので，newメソッドはかならず作る必要があります．また，新しく領域を確保したときに初期化したい項目などがあれば，このnewメソッド内で行えます．しかし，最低限必要な三つの項目を定義しただけですと，VMMとしてはあまりおもしろくありません．

VMMでは，実装しておいたほうがよいメソッド（VMMの「Appendix A」を参照）が定義されています．単体テストを行うためには，まず以下の四つのメソッドが重要です（やることが増えると，作成するべきメソッドも増える）．

- virtual function vmm_data allocate();
- virtual function vmm_data copy (vmm_data to=null);
- virtual function void copy_data (vmm_data to);
- virtual function string psdisplay (string prefix);

各メソッドについて，説明していきます．

1）allocateメソッド

リスト2に示すように，allocateメソッドは領域を確保するメソッドになっています．領域を確保し，その領域のハンドラを返します．

2）copyメソッド

copyメソッドは，以下の二つの使いかたが想定されています．

```
b = a. copy( ) ;
  // 新しい領域bを確保し，bにaの内容をコピーする
a.copy(b);
  // すでにある領域bにaの内容をコピーする
```

このように，引き数を与えないでメソッドを呼び出す場合と，引き数を与えてメソッドを呼び出す場合があります．これを実現する手段がSystemVerilogではデフォルト引き数として用意されています．**リスト2**のcopyの引き数を見ると，vmm_data to=nullとなっていますが，これは呼び出すときに引き数がない場合には引き数toにnullが自動的に設定され，呼び出し時に引き数が与えられた場合には引き数の値がtoに設定されます．

3）copy_dataメソッド

そのcopyメソッドの中で実際に値のコピーを行うのは**リスト3**にあるcopy_dataメソッドになります．このメソッドでは，スーパクラスのデータ・コピーにはスーパク

リスト2 データのクラスのメソッド記述例（その1）

$cast(cp, to)がポイントで，toのハンドルをcpに代入する．これによって正しいデータ型が使われているかを判定する．

```
virtual function vmm_data allocate();
  xy_dat  t = new();        // 新しい領域を確保し，allocateに
  allocate = t;             // その領域のハンドルを渡す
endfunction

virtual function vmm_data copy(vmm_data to=null);
  xy_dat cp;
  if( to==null) cp=new();   // toがnullなら新しい領域を確保する
  else if (!$cast(cp,to) ) begin
    `vmm_error(this.log, "cannot copy");
    copy = null;
    return;
  end
  // それ以外なら，$castを用いてxy_dat型の変数cp
  // にtoのハンドルを渡す．
  // $castでは異なったクラスに代入できない．
  // 代入ができない場合にはエラーを出力し終了させる
  copy_data(cp);            // 現在の変数の内容をcpにコピーする．
  copy = cp;                // 実際にコピーを行うのはcopy_dataメソッド
endfunction
```

リスト3
データのクラスのメソッド記述例（その2）

psdisplayメソッドは，displayメソッドを呼んだときに呼び出されるメソッド．この記述を変更するだけで，さまざまなフォーマットでデータを出力できる．

```
  virtual function void copy_data(vmm_data to);
    xy_dat cp;

    if (!$cast(cp,to)) begin
      `vmm_error(this.log,"cannot copy");
      return;
    end
    super.copy_data(to);
    cp.mX = this.mX;
    cp.mY = this.mY;
  endfunction

  virtual function string psdisplay(string prefix);
    string str, strA, strB;
    strA.itoa(mX);
    strB.itoa(mY);
    str = {prefix, "(", strA, ",", strB, ")"};
    psdisplay = str;
  endfunction

endclass
```

$castでtoのハンドルをcpに代入すると同時に，型のチェックも行う

スーパクラスのcopy_dataメソッドを呼び出す

xy_datのデータ要素に対するコピー処理

mXの文字列にしたものをstring型の変数strAに代入する．これによって，ログなどに出力するメッセージを容易に定義できる

SystemVerilogでは，文字列も連結できる

ラスのcopy_dataメソッドを呼び，自分のクラスの変数はこのメソッドで代入してコピーを行っています．

4）psdisplayメソッド

また，データに関するメッセージを出力するdisplayメソッドがあるのですが，その出力フォーマットを決定するpsdisplayを作成しておきます．これによって，デバッグやログ解析に必要な情報を見やすい形で出力するようにしておくと便利です．もしもこれを作成しないと，スーパクラスのpsdisplayが呼び出されるので，コンパイルそのものは問題なく行えるのですが，今回作成したクラスの変数に関する情報がまったく出力されない，というさみしい状態になります．

● ランダム生成ではrandomizeを呼び出す

データのクラスができたので，ランダム生成を行ってみましょう（下掲のコラム「ランダム生成の威力」を参照）．今回作成したxy_datの変数xを定義し，newによって領域を確保します．その変数xに対してメソッドrandomizeを呼び出します（**リスト4**）．

このrandomizeというメソッドはSystemVerilog

コラム ｜ ランダム生成の威力

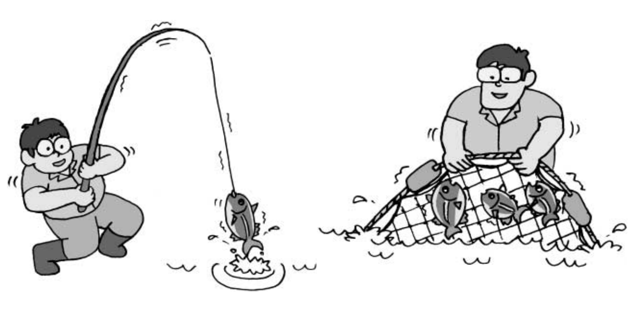
今後の検証ではランダム生成を“うまく”使うことがポイントになります．どんどん複雑化，巨大化する設計に対してランダム生成がどのくらい効果があるのかを，ここでは考えてみます．

今回の設計に対する検証では，1パターンで2個のデータが必要となります．設計者が手動で作成すると1秒間に何パターンを作成することができるでしょうか？ キーボードを打ちながら作成した場合，1秒間に1パターンできればよいほうでしょう．一方，VMMのランダム生成を使うとPentium M（1.1GHz動作）搭載のパソコンでも1秒間に約50万パターンを生成できます．

また，人間はそのペースで1時間作業を続けることができませんが，計算機は24時間でも1年間でも続けることができます．さらに，機械は壊れたら交換できますが，人間が倒れたらそうはいきません．

図A-1に示すように，釣りざおで1匹ずつ魚を釣るのと，漁船の地引き網を使って一気に魚を取る状況に似ています．漁船のほうが効率が良いのはあたりまえですが，船を運転するために免許をとったり，船や網を調達したり，船や網の保守が必要になることを頭に入れておかなければなりません．

うまく使うにはコツがありますが，どう使うと良い検証ができるのかを考えるのは設計者のしごとです（決して，EDAツールのしごとではない）．



図A-1　検証のやりかた

魚を取るにも1本釣りでやる方法もあれば，船から網を投げ入れる方法もある．設計のバグをとる際にもいろいろな方法がある．

リスト4　ランダム生成の実行

ランダム生成を行いたい場合，randomizeメソッドを呼ぶ．このrandomizeメソッドをうまく使うには，VMMが役立つ．

```
program   ex1_randomize;
`include "vmm.sv"

class xy_dat extends vmm_data;
// 省略
endclass                    // xy_dat型の変数xの宣言と領域の確保

initial begin
  xy_dat x=new;
  
  x.randomize();            // randomizeメソッドを呼ぶと，ランダム生成を行う

  x.display();              // xの内容を出力するメソッドをdisplayと呼ぶ
end
endprogram
```

リスト5　チャネルの定義と使いかた

チャネルを使うと，データの受け渡しがメソッドを呼び出すだけになる．

```
class xy_dat extends vmm_data;
 ・・・・
endclass
`vmm_channel(xy_dat)        // チャネルの定義はxy_datを引き数として，マクロを呼ぶだけである．xy_dat用のチャネルはxy_dat_channelとして使用できる

  xy_dat         x, y;
  xy_dat_channel x_chan=new("xy_chan","u0");   // チャネルの領域確保時に，チャネル名とインスタンス名を付ける

initial  begin
  x=new;
  for(int i=0;i<10;i++) begin
      x.randomize();
      #10  x_chan.put(x);   // チャネルx_chanにデータxを投げる
  end
end

initial  begin
  for(int i=0;i<10;i++) begin
      #10  x_chan.get(y);   // チャネルx_chanからデータを受け取る
      y.display( );
  end
end
```

で用意されたものです．このメソッドを呼び出すと，xy_datのmXとmYに対してランダム生成を行います．displayはVMMで用意されたメソッドで，その中からpsdisplayを呼び出してメッセージを出力します．また，新しい値が必要な場合には，再度randomizeを呼ぶことでランダム生成が行えます．VMMではもっと良い方法を使いますが，それは本稿の後半で紹介します．

● データのやり取りにはチャネルを使う

いろいろな部品を使って再利用するためには，あるルールに従って設計を行う必要があります．とくにデータのやり取りを行う通信方式が，再利用のうえで重要なポイントになります．VMMではチャネルvmm_channelを導入することで再利用を促進します．このチャネルを使うと，簡単なインターフェースを使ってデータを送受信できます．また，このチャネルが標準部品なので，拡張するための部品をそのまま接続できます．今回は紹介しませんが，ブロードキャスト（同報）を行う部品も標準ライブラリに用意されています．

リスト5に示すように，チャネルを定義するにはデータ型ごとにマクロを呼びます．そのマクロにより，“データ型_channel”というクラスが作成されます．このチャネルを使うには，“データ型_channel”というクラスの変数を宣言し，実際に領域を確保する必要があります．

チャネルを使うと，データを送信するときはputメソッドを，受信するときはgetメソッドを呼び出すことで，データの送受信が行えます．例えばgetメソッドの場合，データが来るまではgetメソッドは待ちになり，データが来るとそのデータ（正確にはハンドル）を変数に代入します．これは，一つのメソッドで同期待ちの処理と受信処理の二つのしごとを行っていることになり，テストベンチの構築にはたいへん便利です．

● VMMのランダム生成は開始タイミングを制御

先ほどはSystemVerilogの機能を使ってランダム生成を行いましたが，vmm_dataとvmm_channelが使えるようになると，VMMで用意しているランダム生成のしくみを使えます．VMMでは2種類のランダム生成が用意されているのですが，今回は簡単なvmm_atomic_genを使ってみます．

リスト6に示すように，vmm_atomic_genもチャネルと同じようにマクロで定義するだけで簡単にでき上がります．このランダム生成のクラスは出力チャネルを持っています．使いかたは，チャネルの入力側にランダム生成のインスタンスを接続し，ランダム生成を開始するstart_xactorメソッドを呼び出します．このようにランダム生成を開始するタイミングをメソッドで制御できるのが，VMMの一つのメリットになります．また，ランダム生成したデータを使うには，getメソッドにより，チャネルの出力側でデータが出てくるのを待てばよいわけです．

チャネルとランダム生成は，どちらもマクロでデータ型を呼び出すだけで定義でき，さまざまな機能を利用できるすぐれものですが，このマクロ定義には少し注意が必要で

リスト6　atomic_genの定義と使いかた

atomic_genを使うことで検証環境を徐々にレベルアップしていける．

```
program ex3_atomic_gen;
`include "vmm.sv"

class xy_dat extends vmm_data; /*省略*/ endclass

`vmm_channel(xy_dat)
`vmm_atomic_gen(xy_dat, "ATOMIC_GEN")

initial begin
  xy_dat             t;
  xy_dat_channel     g_d_chan=new("XY_channel","u0");
  xy_dat_atomic_gen  xy_gen  =new("u1v,  , g_d_chan);

  fork
    xy_gen.start_xactor();
  join_none

  for(int i=0;i <10; i++) begin
    g_d_chan.get(t);
    t.display();
  end
end

endprogram
```

xy_datのチャネル定義

xy_datのランダム生成を定義．チャネルと同じように，マクロにデータ型と名まえを引き数として与えて呼ぶだけ

ランダム生成のクラスをインスタンス

ランダム生成をスタートする

チャネルからデータを受け取る

リスト7　トランザクタの定義

入力（場合によっては出力）チャネルを定義し，mainのメソッドでデータを受け取って，どのように処理するかを記述する．

```
class xy_rec20 extends vmm_xactor;
  xy_dat_channel    in_chan;

  function new(string instance,
               int stream_id=-1,
               xy_dat_channel in_chan);
    super.new("xy_dat receiver", instance,
stream_id);
    this.in_chan = in_chan;
  endfunction

  virtual task main();
    xy_dat t;
    fork
      super.main();
    join_none

    for (int i=0; i<20; i++) begin
      in_chan.get(t);
      t.display("xy_rec20:");
    end
  endtask

endclass
```

トランザクタの入力となるチャネル変数定義

newしたときに，実際にどのチャネルに接続するかを決定する

vmm_xactorのmainでは，まずスーパクラスのmainメソッドを呼び出す

トランザクタはループ構造になることが多く，データが来るのを待ち，データが来ると処理を開始する

す．通常，SystemVerilogでは文章の終わりにセミコロン（;）を置くのですが，マクロ定義の場合はそのセミコロンが不要です．チャネルやランダム生成のマクロ周辺でエラーが出たら，「セミコロンがない」ことを確認してください．

● ランダム生成はトランザクタの一つの実装

ランダム生成をせっかくVMMの部品で行うことになったのに，受信側はinitialの中でループ文を使って記述しているのはおかしいですね．そこで新しいクラスを紹介します．

受信処理のように何かデータに関する作業を行うものは，VMMではvmm_xactorを使って実現します．まずは，チャネルからデータを受け取って画面に表示する部品を作ってみましょう．**リスト7**にあるように，vmm_xactorから継承し，新しいxy_rec20というクラスを作成します．基本的にはmainメソッドで処理を記述するのですが，トランザクタは一般にループ構造になります．ループの先頭でデータが来るのを待ちます．データを受信したらなんらかの処理を行い，結果を出力し，またループの先頭でデータを待つことになります．

トランザクタのインスタンス時にはnewが呼ばれるので，デフォルト値を持たない引き数，すなわち今回の例ではインスタンス名とチャネルは，インスタンス時に与える必要があります．インスタンスが完了したら，トランザクタを起動します．今回はstart_xactorを呼んで，ランダム生成とトランザクタ生成を実行します．

こうしてみると，ランダム生成とトランザクタが同じように見えます．実はそのとおりで，ランダム生成はトランザクタを使った一つの実装にすぎないのです．このようなところからVMMでは再利用が行われているのです．

● ドライバはインターフェースとトランザクタで構成

検証では，検証対象の回路に対するドライバをこのvmm_xactorを使って作ります．その前に，SystemVerilogのinterfaceについて説明します．

リスト8のように，interfaceはポート信号をまとめてグループとして定義します．今回の設計では，クロック以外の信号を一つのinterfaceとして定義しました．**リスト9(a)**ではポートの代わりにこのまとまりであるinterfaceを使ってprogramを定義しました．**リスト9**

リスト8　インターフェースの定義

ポート信号をひとまとまりにして定義できる．

```
interface judge_if(input bit clk);
  logic       ien;
  logic[7:0]  ix;
  logic[7:0]  iy;
  logic       oen;
  logic       ojudge;
endinterface
```

意味のあるグループでポートをまとめてinterfaceとして定義する

リスト9　インターフェースの使用

インターフェースを使用するとコンパクトに記述できる．また，Verilog 95で記述されたモジュールとは，階層アクセスを使用して接続する．

```
program prog_judge(input clk, judge_if uif);
・・・
```

programの入出力をinterfaceで定義可能

（a）program定義でインターフェースを使った記述

```
module top;
  bit clk;

  judge_if     uif(clk);
  judge_area   udut(clk, uif.ien, uif.ix, uif.iy, uif.oen, uif.ojudge);
  prog_judge   upg(clk, uif);

  initial begin
      #20  clk = 1'b0;
    forever begin
      #50  clk = ~clk;
    end
  end
endmodule
```

interfaceのインスタンスを定義

DUTに対してはinterfaceの階層アクセスを使ってインスタンス

programにはinterfaceを使ってインスタンス

（b）トップ階層でインスタンスにインターフェースを使った記述

(b)のトップ・モジュールでは，インターフェース自身もインスタンスし，そのインターフェースを使ってDUT（design under test；検証対象となる設計）とprogramを接続します．DUTはVerilog 95（IEEE 1364-1995）の信号ごとにポート定義しているので，インターフェースと接続するためにはインターフェースの階層アクセスを使って個別信号でDUTのインスタンスを行います．

設計に対して入力データを渡す処理を行うドライバは，このインターフェースとトランザクタを使って構成することができます．今回は，ランダム生成からデータを受け取る部分はチャネルと接続し，DUTとの接続にはアクセス・メソッドを使います（**リスト10**）．

ドライバは，ランダム生成からデータを受け取ったら，クロックの立ち上がりから10ほど遅延させ，データを出力すると同時に制御信号ienを‘1’にします．その次のクロックの立ち上がりから10遅延で，制御信号ienを‘0’にします．また，データの出力も‘0’にします（これにはclockingを使いたいところだが，紹介はまた別の機会に…）．

● vmm_envで呼び出し順序を定義

これまでのところでVMMの決まりに従ってテストベンチを作成してきました．これまでの部分でもランダム生成やドライバは再利用が可能になります．例えば，設計の入力インターフェースが変わった場合でも，ドライバを変更するだけで対応できます．逆にドライバはそのままで，異なったシミュレーション・パターンを生成するためにランダム生成だけを変更することもできます．

その再利用性を向上する最後の部品がvmm_envです．これによりテストベンチでのインスタンスや部品の呼び出し順序を規定しています．再利用するうえで困ることとして，

リスト10　ドライバをトランザクタで作成

mainメソッドのループでデータを受信したら，インターフェース信号をドライブするdut_driveを呼び出す．

```
class xy_drive20 extends vmm_xactor;
  xy_dat_channel  in_chan;

  function new(・・・); /* 省略 */ endfunction

  virtual task main();
    xy_dat  t;
    fork
      super.main();
    join_none
    for (int i=0; i<20; i++) begin
      in_chan.get(t);
      dut_drive(t);
    end
  endtask

  virtual task dut_drive(xy_dat t);
    @(posedge clk)
      #10 uif.ien <= 1'b1;
          uif.ix  <= t.mX;
          uif.iy  <= t.mY;
    @(posedge clk)
      #10 uif.ien <= 1'b0;
          uif.ix  <= 0;
          uif.iy  <= 0;
  endtask
endclass
```

データを受信したら，dut_drive メソッドを呼び出す

データを受信したら，dut_drive メソッドを呼び出す

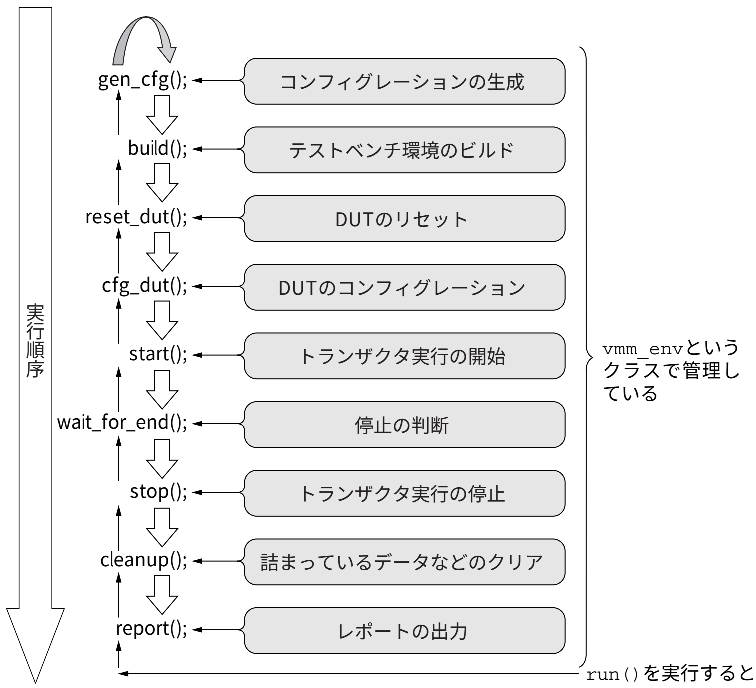


図3　VMMの実行ステップ

gen_cfgメソッドからreportメソッドまで順に実行される．ユーザが作ったものは，メソッドを拡張して組み込むことができる．

リスト11　vmm_envの作成

今回は，buildでチャネル，ランダム生成，ドライバのインスタンスを作成し，startでランダム生成とドライバを起動し，stopでランダム生成とドライバを停止した．

```
class xy_env extends vmm_env;
  xy_dat_channel      g_d_chan;
  xy_dat_atomic_gen   xy_gen;
  xy_drive20          xy_drv;

  virtual function void build();
    super.build();
   g_d_chan=new("Gen_drv_channel","u0");
    xy_gen  =new("u1",,g_d_chan);
    xy_drv = new("u2",,g_d_chan);
  endfunction

  virtual task start();
    super.start();
      xy_gen.start_xactor();
      xy_drv.start_xactor();
  endtask

  virtual task stop();
    super.stop();
      xy_gen.stop_xactor();
      xy_drv.stop_xactor();
  endtask
  /*   省略  */
endclass
```

呼び出し順序によって結果が変わってしまうことがあります．そのため，VMMでは**図3**のように呼び出されるメソッドと役割が決められています．例えば，先ほどのトランザクタでは，インスタンスはbuildメソッドで行いますし，トランザクタの起動はstartメソッドで行い，トランザクタの停止はstopメソッドで行います．

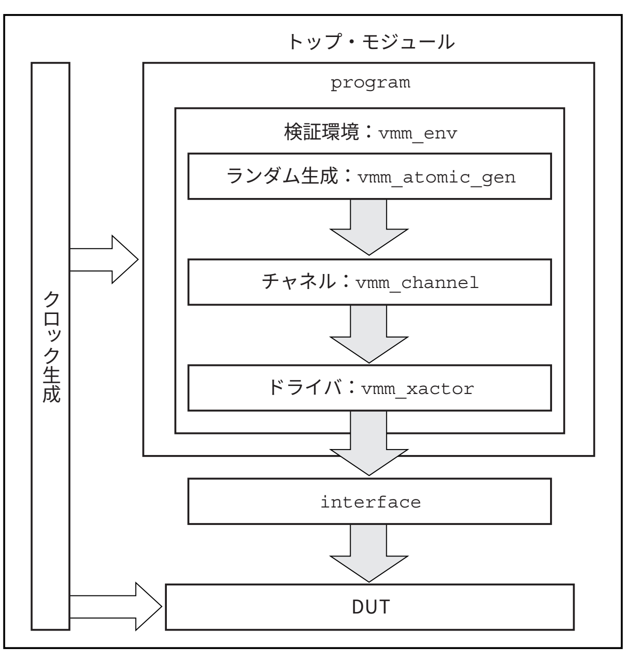


図4　トップ・モジュールの構成

トップ・モジュールには，programやinterface，DUT，クロック生成がインスタンスされる．

リスト11に，build，start，stopについて拡張したメソッドを示しています．拡張する場合は，まずスーパクラスのメソッドを呼んでから，今回の検証で必要なメソッドを呼び出します．buildではチャネル，ランダム生成，ドライバをインスタンスし，startではランダム生成とドライバを起動し，stopではランダム生成とドライバを停止します．

テストベンチとしては，programの中で次の4行の記述が必要です．

```
initial begin
 xy_env env=new;
 env.run();
end
```

この記述は，検証環境をインスタンスし，runメソッドを呼んで実行するだけです．このrunメソッドを呼ぶと，**図3**のステップを上から実行します．ランダム生成した結果に対してドライバにより制御信号を付加し，ターゲット

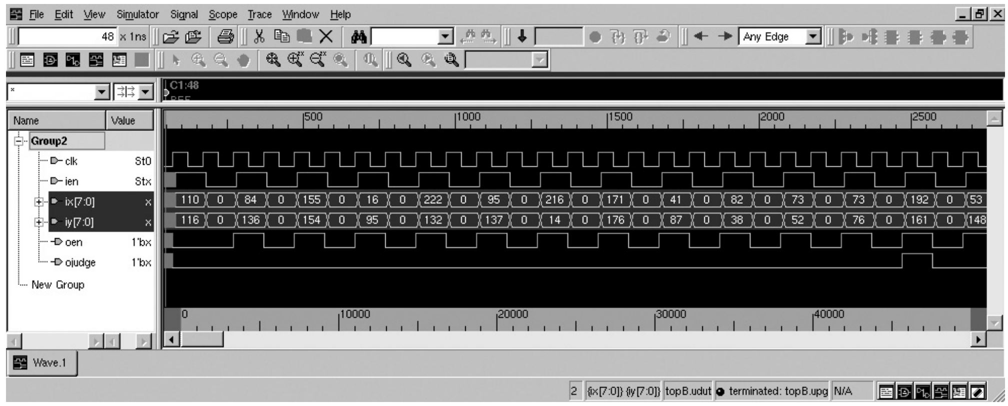


図5
シミュレーション波形
明示的に入力値を決めなくても，ランダム生成によってシミュレーションを進めていくことが可能となる．とはいっても，ランダム生成で検証のすべての問題が片づくほど世の中は甘くないのだが….

回路に入力として与えることができます．

● **トップ階層やクロックを記述してテストベンチが完成**

これでテストベンチ側が完成したので，**図4**にトップ・モジュールの構成を示します．**リスト9(b)**に示すトップ・モジュールの記述では，インターフェースとテストベンチであるprogram，検証対象である設計をインスタンスし，クロックを生成させています．

クロック生成についてもいくつかのVMMのルールがあり，「Rule 4-15」に従ってトップ・モジュールでクロックを生成し，また「Rule 4-16」に「時刻0ではクロックを発生させない」と述べられていることから，20遅延ほどずらしてクロック信号を変化させています．

これで，実際にシミュレーションを動作させると，ランダム生成によって検証が進み，**図5**に示した波形が出てきます．

今回のテストベンチで注目していただきたいのは，*X*，*Y*座標については数値を直接指定した部分がない，という点です．

検証の終わりを，実現可能なすべての組み合わせのパターーンの入力とするならば，人手でそのすべてのパターンを書くことは不可能です．そんなに話を大きくしなくても，32ビット加算器のパターンですら，スクリプトやプログラムを使用しないで人手で書くことはできないでしょう．そのため，今後の検証ではランダム生成をうまく活用することが重要になります．次回以降にまた詳しく説明しますが，VMMの観点では，直接数値を記述するダイレクトなテストベンチは，ランダム生成の一つのバリエーションに過ぎません．VMMでは，ダイレクトなテストベンチとランダムなテストベンチが共存できる点がおもしろいところです．

参考・引用*文献

(1) * Janick Bergeron，Eduard Cerny，Alan Hunter，Andrew Nightingale；Verification Methodology Manual for System Verilog，Springer，2005（邦訳は，ベリフィケーション・メソドロジ・マニュアル，CQ出版社，2006年）．
(2) 設計者のためのSystemVerilogお役立ち情報，http://www.eda-express.com/sp/SystemVerilog/（programに関する資料がある）．

あかぼし・ひろき
(株)ロジック・リサーチ

<筆者プロフィール>
赤星博輝．ハードウェアからソフトウェアまで担当するなんでも屋．現在はディジタル検証ではVerification Methodology ManualとSystemVerilogを推進し，ハードウェア検証のためにリアルタイムOS技術にも進出．